Umelá inteligencia

Zadanie č.1 - Prehľadávanie stavového priestoru

(8 hlavolam – problém 2d)

Autor: Marek Čederle
AIS ID: 121193 (xcederlem)
Cvičiaci: Ing. Nina Masaryková
(Streda 15:00)

Úvod

Ako spustiť program (Linux/Unix like)

Je potrebné aby sme mali súbory main.cpp a tree.h v jednom adresáry. Otvoríme si príkazový riadok v adresári so súbormi a skompilujeme program pomocou kompilátora g++. Zadáme nasledujúce príkazy:
> g++ main.cpp -o main
> ./main

Po spustení program vypíše tieto inštrukcie pre pokračovanie a informácie o použitej heuristickej funkcii:

```
Misplaced heuristic:
Matrix 3x3:
Number '0' represents the blank space
Example:
1 2 3
4 5 6
7 8 0
Enter the initial state of the puzzle:
```

Opis problému

Úlohou tohto zadania bolo vyriešenie známeho 8 hlavolamu s tým, že cieľový stav nemusí byť postuponosť čísel zoradená vzostupne. Máme 9 políčok pričom na 8 políčkach sú štvorčeky s číslami a jedno prázne políčko (bez štvorčeka). Našou úlohou je nájsť postuponosť krokov zo zadaného počiatočného stavu do zadaného cieľového stavu. Máme možnosti pohybu HORE, DOLE, VĽAVO, VPRAVO a pomocou nich sa snažíme dosiahnuť riešenie.

Implementácia

Na implementáciu som použil jazyk C++ a nasledovné programy:

- Vývojové prostredie (IDE) – Clion
- Kompilátor – g++
- Kontrola pamäti – valgrind (program je bez únikov pamäti)
- Grafy a tabuľky – Google Sheets
- Dokumentácia – LibreOffice Writer

Opis riešenia

Definícia uzla:

```
class Node{
public:
    int state[N]{};
    int depth;
    direction dir;

    Node *up;
    Node *down;
    Node *left;
    Node *right;
    Node *prev;
```

Tento ústrižok kódu definuje triedu Node (Uzol) s nasledujúcimi dôležitými premennými:

- state – pole celých čísel, reprezentuje zadaný stavového
- depth – hĺbka daného uzla v strome
- dir – aká operácia (smer) bol použitý na dostanie sa do tohto stavu
- up, down, left, right, prev – ukazovateľe na nasledujúce potenciálne stavy a predchádzajúci stav

Obsahuje aj konštruktor a deštruktor ktorý zabezpečujú úvodné priradenie dát a následné zmazanie objektu.

Definícia stromu:

```
class Tree{
public:
    Node *root;
    std::vector<Node*> created;
    std::unordered_map<string, Node*> processed;
```

Tento kód definuje triedu Tree (Strom)s nasledujúcimi premennými a funkciami:

- root – prvý uzol stromu (koreň)
- created – vektor vytvorených stavov
- processed – hashovacia mapa procesovaných stavov

Heuristická funkcia na zistenie počtu políčok, ktoré nie sú na svojom mieste:

```
static int misplacedSquares(const int state[], const int goal[])
{
    int sum = 0;
    for (int i = 0; i < N; i++)
    {
        if (state[i] != goal[i])
        {
            sum += 1;
        }
    }
    return sum;
}
```

Fukcia porovnáva či sa na rovnakých indexoch nachádza rovnaké číslo aj pre aktuálny stav aj pre cieľový a vráti súčet počtu prvkov, ktoré nie sú na svojom mieste.

Heuristická funkcia na zistenie súčtu vzdialeností jednotlivých políčok od cieľovej pozície:

```
static int manhattanDistance(const int state[], const int goal[])
{
    int sum = 0;
    for(int i=0;i<N;i++)
    {
        int temp_index = getIndex(state,  value: goal[i]);
        int x = temp_index % 3;
        int y = temp_index / 3;
        int x_goal = i % 3;
        int y_goal = i / 3;
        sum += abs( x: x - x_goal) + abs( x: y - y_goal);
    }
    return sum;
}
```

Funkcia nájde na akom indexe v aktuálnom stave (state) sa nachádza číslo z daného indexu z cieľového stavu. Následne z toho zistí súradnice tohto čísla v oboch stavoch (aktuálny, cieľový) a vypočíta dĺžku vzdialenosti medzi nimi.

Trieda Tree (Strom) obsahuje aj iné pomocné funkcie, ktorých zmysel je zrejmý z kódu programu. Obsahuje aj iné dôležité funkcie ktoré budú ale vysvetlené spolu s celkovým fungovaním programu a algoritmu.

Pri spustení programu (teda spustení funkcie main) dôjde ku zavolaniu funkcie compute, ktorá má na starosti všetko čo sa týka čítania zo vstupu, vytvorenia štruktúr a implementovania algoritmu.

Compute funkcia a fungovanie algoritmu:

V úvode funkcia vypíše úvodné inštrukcie a načíta iniciálny a cieľový stav. Potom vytvorý objekty dátových štruktúr. Následne zistí pomocou funckie checkFinal či už iniciálny stav je zároveň cieľový, Ak áno tak skončí program s tým, že hlavolam je už vyriešený. Ak nie pokračuje v behu programu a zistí či je tento hlavolam vôbec riešiteľný pomocou funkcie solvable.

```
static bool solvable(int state[], int goal[])
{
    int sum_goal = 0;

    for(int i=0;i<N;i++)
    {
        if (state[i] == 0) continue;
        for(int j=i+1;j<N;j++)
        {
            if (state[j] == 0) continue;
            // this if-clause detects if the number is not in the array
            if(getIndex( state: goal,  value: state[i]) == -1 || getIndex( state: goal,  value: state[j]) == -1)
            {
                return false;
            }

            if(getIndex( state: goal,  value: state[i]) > getIndex( state: goal,  value: state[j]))
            {
                sum_goal++;
            }
        }
    }

    return !(sum_goal % 2);
}
```

Algoritmus na zistenie či je hlavolam riešiteľný funguje nasledovne. Začíname od nultého indexu a zoberieme si prvé číslo a pozeráme sa vľavo, či sa tam nachádza nejaké číslo ktoré je väčšie ako dané číslo a zistujeme koľko ich je. Toto robíme zároveň pre iniciálny stav a robíme to zároveň aj pre náš cieľový stav. Takto postupujeme po všetkých indexoch a na konci nám vyjde súčet týchto čísel. Na konci ku našim súčtom pripočítame v ktorom riadku sa nachádza medzera. Následne urobíme modulo dvomi na obe čísla a výsledky z iniciálneho stavu a cieľového stavu porovnáme ak sa rovnajú tak je hlavolam riešiteľný ak nie tak nie je riešiteľný. Jediný rozdiel v tomto algoritme a mojej implementácii je, že neporovnávame priamo čísla ale indexy tých čisel z dôvodu že náš cieľový stav nemusí byť presne usporiadaný rad čísel vzostupne.

Po prebehnutí tejto funkcie sa pridá koreň stromu do vektora vytvorených stavov a začína nám cyklus v ktorom sa budeme točiť pokým neprídeme ku riešeniu.

V cykle to funguje nasledovne:

```
int temp_heuristic=Tree::applyHeuristic(tree->created[0]->state, goal,  heuristic: heuristic_pass);
int temp_index=0;

for(int i=0; i< tree->created.size(); i++)
{
    if(Tree::applyHeuristic(tree->created[i]->state, goal,  heuristic: heuristic_pass) < temp_heuristic)
    {
        temp_heuristic = Tree::applyHeuristic(tree->created[i]->state, goal,  heuristic: heuristic_pass);
        temp_index = i;
    }
}

current = tree->created[temp_index];
```

Pomocou lačného vyhľadávania t.z. aplikovaní nami určenej heuristickej funckie a vybratí najmenšieho čísla si z vytvorených stavov, ktoré neboli procesované vyberieme potenciálne najlepší smer kadiaľ sa vydáme a nastavíme si ho ako aktuálny (current) stav, cez ktorý budeme následne iterovať.

```
if(Tree::checkFinal( node: current, goal))
{
    final = current;
    break;
}

tree->generateStates( node: current);
tree->created.erase( first: tree->created.begin() + temp_index,  last: tree->created.begin() + temp_index + 1);
```

Následne skontrolujeme či už sme nenašli finálny stav. Ak nie, tak vygenerujeme možné stavy a vymažeme aktuálny stav ktorý sme procesovali z vytvorených a neprocesovaných a ideme späť na začiatok cyklu.

Generovanie stavov prebieha nasledovne:

```
void generateStates(Node *node)
{
    std::vector<direction> states = possibleStates(node);

    if(states[0] == ERROR)
    {
        return;
    }

    for(int i=0;i<states.size();i++)
    {
        int new_state[N];
        for(int j=0;j<N;j++)
        {
            new_state[j] = node->state[j];
        }
        switch (states[i])
        {
            case UP:
                swap( state: new_state,  index1: getIndex( state: new_state,  value: 0),  index2: getIndex( state: new_state,  value: 0) - 3);
                break;
            case DOWN:
                swap( state: new_state,  index1: getIndex( state: new_state,  value: 0),  index2: getIndex( state: new_state,  value: 0) + 3);
                break;
            case LEFT:
                swap( state: new_state,  index1: getIndex( state: new_state,  value: 0),  index2: getIndex( state: new_state,  value: 0) - 1);
                break;
            case RIGHT:
                swap( state: new_state,  index1: getIndex( state: new_state,  value: 0),  index2: getIndex( state: new_state,  value: 0) + 1);
                break;
            default:
                break;
        }
```

Vytvoríme si vektor možných stavov pomocou funkcie possibleStates. Táto funkcia iba zistí na akom indexe sa nachádza medzera a podľa toho nám vráti vektor možných operácií.
Následne ošetrujeme či nenastala chyba, ak nie tak iterujeme cez tento vektor a vytvoríme pre každú novú operáciu nový stav kde vymeníme podľa operácie pozíciu medzery.

```
        if(processed.find( x: getStrRepresentation( arr: new_state)) == processed.end())
        {
            Node *new_node = new Node( state: new_state,  depth: node->depth + 1);
            new_node->prev = node;
            new_node->dir = states[i];

            switch (states[i])
            {
                case UP:
                    node->up = new_node;
                    break;
                case DOWN:
                    node->down = new_node;
                    break;
                case LEFT:
                    node->left = new_node;
                    break;
                case RIGHT:
                    node->right = new_node;
                    break;
                default:
                    break;
            }
            created.push_back(new_node);
            processed[getStrRepresentation( arr: new_state)] = new_node;
        }
    }
}
```

Vo funkcii pokračujeme takto:

Ak v hashovacej mape procesovaných stavov nenajdeme tento náš nový stav (toto ošetruje generáciu tých istých stavov a následného cyklenia sa), tak potom ho ideme procesovať. To znamená že podľa toho ktorým smerom ideme (ktorú operáciu sme použili) tak nastavíme, že tento nový stav sme získali z danej operácie a zároveň nastavíme novému stavu predchodcu čiže akutúlny stav. Nakoniec pridáme nový stav medzi vytvorené stavy.

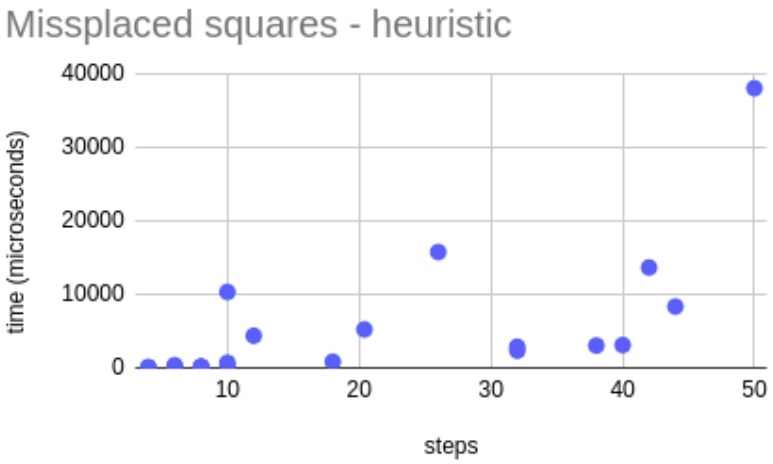
Po celom behu hlavného cyklu na konci vypíšeme aký bol pôvodný a cieľový stav a koľko krokov respektíve ako hlboko je náš uzoľ, v ktorom je cieľový stav.

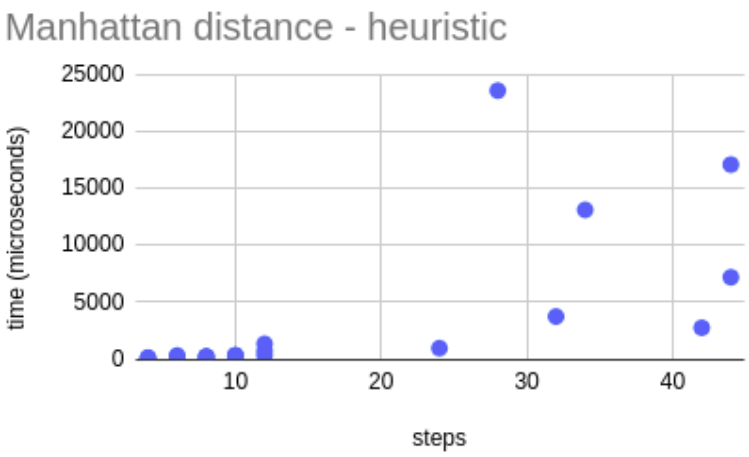
Na záver pomocou spätého hľadania vypíšeme postupnosť krokov ako sme sa ku riešeniu dostali.

Testovanie a záver

Svoju implementáciu som testoval pomocou 20 rôznych počiatočných stavov a toho istého cieľového a následne som tieto isté stavy vymenil medzi sebou.

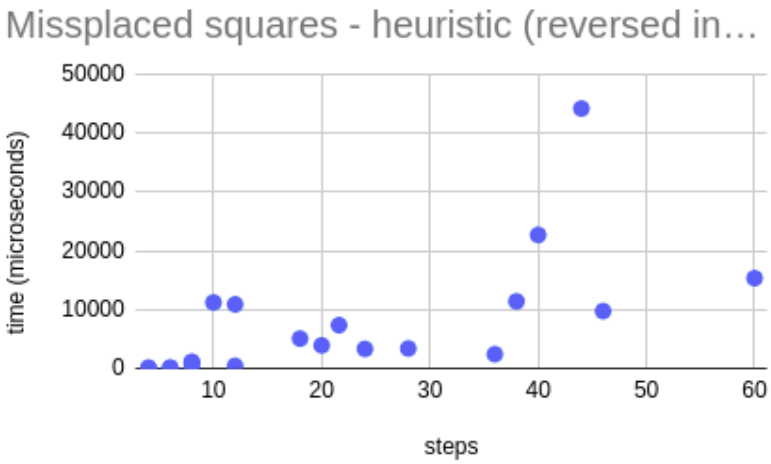
			steps		time taken (microseconds)			missplaced		manhattan	
ID	init state	goal state (same for all)	missplaced squares	manhattan	missplaced squares	manhattan		steps	time (microseconds)	steps	time (microseconds)
1	{6, 2, 0, 1, 5, 3, 4, 7, 8};	123456780	42	12	13662	714		42	13662	12	714
2	{1, 2, 3, 4, 6, 8, 7, 5, 0};	123456780	4	4	132	79		4	132	4	79
3	{7, 3, 5, 2, 4, 1, 8, 6, 0};	123456780	32	44	2390	7195		32	2390	44	7195
4	{2, 3, 8, 7, 1, 5, 0, 4, 6};	123456780	32	24	2917	950		32	2917	24	950
5	{2, 4, 3, 1, 6, 8, 0, 7, 5};	123456780	10	10	194	298		10	194	10	298
6	{4, 1, 6, 7, 3, 2, 0, 5, 8};	123456780	10	10	755	326		10	755	10	326
7	{1, 2, 3, 7, 4, 8, 0, 6, 5};	123456780	8	8	195	174		8	195	8	174
8	{1, 3, 0, 4, 2, 8, 7, 6, 5};	123456780	8	8	204	193		8	204	8	193
9	{1, 2, 3, 8, 0, 5, 4, 7, 6};	123456780	6	6	409	219		6	409	6	219
10	{4, 2, 3, 7, 1, 8, 5, 6, 0};	123456780	40	28	3154	23634		40	3154	28	23634
11	{1, 3, 5, 4, 8, 2, 7, 6, 0};	123456780	8	8	212	223		8	212	8	223
12	{1, 7, 2, 5, 0, 3, 6, 4, 8};	123456780	50	42	37982	2750		50	37982	42	2750
13	{0, 1, 3, 4, 2, 5, 7, 8, 6};	123456780	4	4	149	120		4	149	4	120
14	{4, 1, 2, 7, 0, 3, 6, 8, 5};	123456780	44	12	8371	319		44	8371	12	319
15	{4, 2, 3, 5, 1, 6, 0, 7, 8};	123456780	12	12	4417	1334		12	4417	12	1334
16	{4, 1, 3, 2, 0, 6, 7, 5, 8};	123456780	6	6	197	305		6	197	6	305
17	{0, 2, 6, 1, 3, 8, 7, 4, 5};	123456780	26	32	15766	3736		26	15766	32	3736
18	{1, 3, 6, 2, 0, 4, 7, 8, 5};	123456780	38	34	3088	13132		38	3088	34	13132
19	{2, 3, 5, 7, 1, 4, 0, 8, 6};	123456780	10	10	10350	290		10	10350	10	290
20	{3, 6, 8, 1, 0, 2, 5, 4, 7};	123456780	18	44	883	17118		18	883	44	17118
							avg:	20,4	5271,35	17,9	3655,45

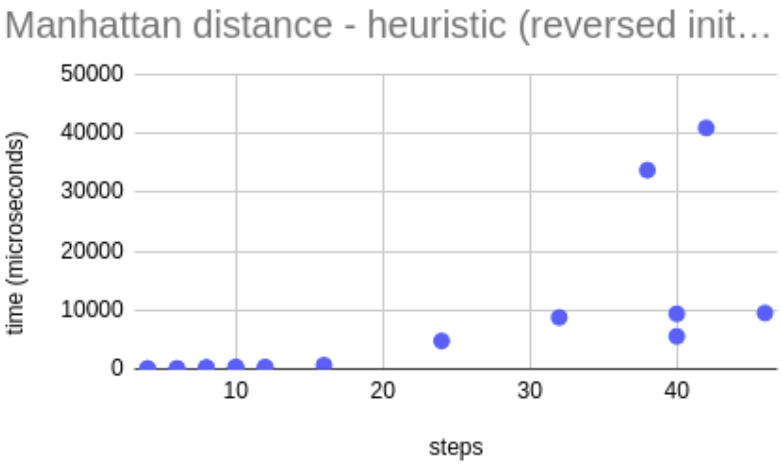




Prehodené iniciálne a cieľové stavy:

	inverted from previous table		steps		time taken (microseconds)			missplaced		manhattan	
ID	goal state	init state (same for all)	missplaced squares	manhattan	missplaced squares	manhattan		steps	time (microseconds)	steps	time(microseconds)
1	{6, 2, 0, 1, 5, 3, 4, 7, 8};	123456780	12	12	496	341		12	496	12	341
2	{1, 2, 3, 4, 6, 8, 7, 5, 0};	123456780	4	4	163	135		4	163	4	135
3	{7, 3, 5, 2, 4, 1, 8, 6, 0};	123456780	46	46	9779	9546		46	9779	46	9546
4	{2, 3, 8, 7, 1, 5, 0, 4, 6};	123456780	44	40	44121	5593		44	44121	40	5593
5	{2, 4, 3, 1, 6, 8, 0, 7, 5};	123456780	40	10	22693	188		40	22693	10	188
6	{4, 1, 6, 7, 3, 2, 0, 5, 8};	123456780	28	10	3442	415		28	3442	10	415
7	{1, 2, 3, 7, 4, 8, 0, 6, 5};	123456780	8	8	1076	271		8	1076	8	271
8	{1, 3, 0, 4, 2, 8, 7, 6, 5};	123456780	8	8	1188	212		8	1188	8	212
9	{1, 2, 3, 8, 0, 5, 4, 7, 6};	123456780	6	6	183	102		6	183	6	102
10	{4, 2, 3, 7, 1, 8, 5, 6, 0};	123456780	38	40	11435	9404		38	11435	40	9404
11	{1, 3, 5, 4, 8, 2, 7, 6, 0};	123456780	8	8	510	273		8	510	8	273
12	{1, 7, 2, 5, 0, 3, 6, 4, 8};	123456780	24	42	3369	40875		24	3369	42	40875
13	{0, 1, 3, 4, 2, 5, 7, 8, 6};	123456780	4	4	148	133		4	148	4	133
14	{4, 1, 2, 7, 0, 3, 6, 8, 5};	123456780	18	24	5120	4822		18	5120	24	4822
15	{4, 2, 3, 5, 1, 6, 0, 7, 8};	123456780	36	16	2469	718		36	2469	16	718
16	{4, 1, 3, 2, 0, 6, 7, 5, 8};	123456780	6	6	170	153		6	170	6	153
17	{0, 2, 6, 1, 3, 8, 7, 4, 5};	123456780	12	12	10920	351		12	10920	12	351
18	{1, 3, 6, 2, 0, 4, 7, 8, 5};	123456780	20	32	3969	8787		20	3969	32	8787
19	{2, 3, 5, 7, 1, 4, 0, 8, 6};	123456780	10	10	11239	234		10	11239	10	234
20	{3, 6, 8, 1, 0, 2, 5, 4, 7};	123456780	60	38	15362	33726		60	15362	38	33726
							avg:	21,6	7392,6	18,8	5813,95





Z týchto tabuliek a grafov vyplýva, že heuristická funkcia na zistenie súčtu vzdialeností jednotlivých políčok od cieľovej pozície (manhattan distance) je v priemere o niečo lepšia ako heuristická funkcia ktorá počíta koľko políčok nie je na svojom mieste (missplaced square). Avšak je dôležité povedať že vždy to závisí od zadaných stavov a dokonca aj ich poradí ako je možné vidieť v tabuľke. Ak sa pozreme na dľžku času, tak aj ona nám potvrdzuje toto tvrdenie. Samozrejme nie je možné toto hodnotenie úplne generalizovať pretože záleží od hardwarovej (HW) konfigurácie zariadenia a v niektorých prípadoch aj vývojového prostredia.

Zdroje

http://www2.fiit.stuba.sk/~kapustik/ui.html
https://en.wikipedia.org/wiki/Greedy_algorithm
Prednášky z predmetu UI (Umelá inteligencia)
Prednášky z predmetu AZA (Analýza a zložitosť algoritmov)